

Control-Restrictive Instructions for Structured Programming (CRISP)

R. C. Tausworthe

DSN Data Systems Development Section

This paper presents a discipline and a set of control-logic statements to extend structured programming to arbitrary existing languages. These statements preempt and replace all control statements in a language, so that all programs written in Control-Restrictive Instructions for Structural Programming (CRISP) are automatically structured. Structures are provided for real-time, as well as nonreal-time programming. The principles set forth do not attempt to specify a standard programming language, but instead, a programming language standard—that is, a way of programming that contributes to stability, maintainability, readability (self-documentation), and understandability of the final product.

I. Introduction

The purpose of a higher-level programming language has historically been to simplify the expression of algorithms or subprogram functions created by an important class of problems. The flexibility and productivity of such languages are gauged by the ease with which, and the degree to which programmers may vary the composition and execution of programs (Ref. 1). The widely diverse classes of problems that exist have, over the years, led to the development of an exceedingly large number of languages, both wide-application (general-purpose) and restricted-application (special-purpose). There is no doubt that standardization is needed, but defining a “standard

language” is probably only feasible within a distinct problem class.

The characteristics sought in a standard language, however, are noble: the language should be capable of solving problems over a wide range of applicability, and contribute to the solution of those problems large measures of stability, maintainability, readability (or self-documentation), understandability, and machine (or installation) independence. Furthermore, it should lend itself as much as possible to program production tools, automatic design methods, easy assessment of correctness, easy or automated verification and testing, and easy or

automated quality assurance measures. To be acceptable, as a *minimum* requirement, a standard computer language must not hinder the programming process. On the contrary, the purpose of a standard is to help.

The principles set forth in this work do not attempt to specify a standard programming language, but instead, do provide a *programming language standard*—that is, a disciplined *way* of programming to achieve the goals of the preceding paragraph.

Restricted control-logic structures, as proposed by Böhm and Jacopini (Ref. 2) and others (Refs. 3 and 4), and extended by the author (Ref. 5) form the basis of an attractive software design and production methodology known as “structured programming.” Programs written using these restricted control-logic structures, tend to be compatible with and enhance other widely useful techniques, such as top-down methods (Ref. 6), modular programming, and hierarchic design (Ref. 7). Such programs are found to be easier to organize, understand, modify, and manage.

In block-structured programming languages, such as ALGOL and PL/I, structured programs are GOTO-free. Structured programming, however, can be extended to almost any language, and should not be characterized simply by the absence of GOTO's, but rather by the presence of an organized control-logic discipline.

II. The CRISP Concept

Program control-logic is specified in what follows here by way of a set of Control-Restrictive Instructions for Structured Programming, called CRISP, augmenting an arbitrary target language. Programmers construct code using statements from the arbitrary target language, such as FORTRAN, BASIC, or assembly language, except for statements governing the program control-logic (branching, looping, etc.): such control is accomplished by using a CRISP statement instead.

The source-program statements thus consist of a mixture of CRISP and target-language code, which can then be processed into executable instructions for a given computer system. The processor may take the form of a compiler, by which the source statements are translated directly into executable form; but rewriting or modifying an existing compiler to accommodate CRISP can be averted by implementing the translation via a CRISP preprocessor. Neither exists at this writing.

Such a CRISP preprocessor would access the sequential source records, written in CRISP or target-language syntax and replace the control-logic statements by target-language statements that perform the equivalent action.

The CRISP control structures are precisely those needed to write structured programs—even nonreal-time structured programs (Ref. 8). The CRISP concept thus extends the advantages of structured programming to those languages that most fit a particular problem.

CRISP preempts all control statements from the target language and substitutes a set of statements that force programs to be structured; that is, any program written in CRISP is automatically structured without the need for GOTOs. “GOTO-less” structured programming is currently available in some other languages now coming into being, such as BLISS (Ref. 9), IFTRAN (Ref. 10), and SIMPL-X (Ref. 11); a special limited preprocessor for FORTRAN, called SFTRAN (Ref. 12) is also now available.

The strength of CRISP, as opposed to these other structured programming languages lies in the fact that *only* the control statements are preempted. Given an operating CRISP preprocessor for the target language most suitable for the problem at hand, the user may proceed to solve the problem in the language he wants, and is already familiar with. If he is called upon to solve another problem in another familiar language, then he again finds the same set of control-logic statements by which to organize that problem in the other language.

III. Elements of CRISP Statements

A CRISP statement begins with a reserved word identifying the type of structure, or the module within a structure, or the end of a structure. Additionally, CRISP statements may contain strings that belong to the target language, or other CRISP statements. For example, in the CRISP structure of Fig. 1, the substring denoted by *c* is a condition that will be substituted directly into a conditional statement in the target language so as to produce the structure shown in Fig. 1. The strings *s_i* in Fig. 1 are either target statements or other nested CRISP constructions.

The complete superset of CRISP constructions is given in the Appendix, along with flowchart equivalents. (Not all of these will apply to a given target language.) Each program structure will be here referred to as a *CRISP-block* (not to be confused with the definition of a block in

block-structured languages such as ALGOL and PL/I); subdivisions of blocks into constituent parts will be referred to as *modules*. Blocks and modules will be typed by their initiating key-words, as for example, an UNLESS-block, or a THEN-module; in some cases, block names may need further description, such as may be desirable to contrast an IF-THEN-ELSE block from an IF-THEN block, or a FOR-UNTIL block from a FOR-WHILE block.

The CRISP structures can conceptually be iterated and nested to any level desired to produce the intended program. Indentations and annotations for readability (which I shall discuss later), however, will tend to limit the amount of nesting within blocks, because the listing tends to crowd toward the right-hand edge. Rather than contend with this continued crowding, the user naturally finds himself inventing procedures to be linked or called (and programmed later). As a result, CRISP programs, subprograms, and subroutines generally fit on one page each (but link to procedures on other pages).

As Mills (Ref. 4) points out, segmentation of program listings to a predescribed size, such that each segment enters only at the top and exits (normally) at the bottom, is a major asset in coping with program complexity.

CRISP makes allowance for up to three distinct types of procedure calls within a program. The first is of the form

DO *f*

which links to the procedure named *f* in a

PROCEDURE: *f*

.
.
.
END

block. In some CRISP processors, it is conceivable that the entire procedure *f* could be substituted for the DO *f* statement in the object code. Arguments may sometimes be passed in the calling string *f*.

The second procedure call is

CALL *f*

which creates a subroutine linkage to a named procedure declared in a

SUBROUTINE: *f*

.
.
.
RETURN

block. Subroutine arguments may be passed in the normal way between CALL and the SUBROUTINE: definition.

The third procedure call is

GOSUB *l*

where *l* is a subroutine label; arguments are not generally passed, except in the form of common variables, tables, and so forth. The labeled subroutine is defined in a block of the form

SUBROUTINE LABEL: *l*

.
.
.
RETURN

The two different types of subroutine calls are necessary in target languages (Ref. 13) that are capable of calling subroutines both by name and by label. One or the other may be absent in other languages, however.

Functions, when permitted in the target language, are identified by block declarations of the form

FUNCTION:

.
.
.
RETURN *answer*

The *answer* string is an optional device that may be required in some target languages to return the function value. Functions are invoked in the usual target-language mode.

The main program is identified as the block

PROGRAM: *name*

.
.
.
SYSTEM (*or* STOP)

The directive SYSTEM releases the control of execution to the system; STOP, to the operator. Again, both of

these options may not be available in an arbitrary target language.

IV. A CRISP Preprocessor

Because the CRISP statements are keyword-actuated, it is necessary that target-language noncontrol statements not begin with these keywords. Otherwise, alternate CRISP keywords must be chosen. More detailed restrictions will appear later in this article. The processor functions will only be outlined here.

The CRISP preprocessor functions in a number of modes, and I will describe aspects of each in turn. The main mode is the *translation* mode, which outputs target-code statements. The second and third modes are edit modes: *update* and *annotation*. The update mode is a text-editor that permits insertions, deletions, and alterations of CRISP programs; the annotation mode indents CRISP blocks and supplies them with flowlines and Dewey-decimal reference and cross-reference numbers.

The processor allows comments to appear anywhere in a program, within target-language statements, as well as within CRISP control statements, and are indicated¹ by surrounding the comment string by "<*" and ">", as, for example, by <* comment >. The *comment* may then contain any string of characters except ">". CRISP comments continue automatically on the next line if they are incomplete on the current line.

The strings "<*" and ">" naturally, must not be permitted constructs in target language statements. If either is, an alternate comment delimiter may be substituted as a convention for implementing CRISP in that target language.

CRISP statements may be continued on several lines by terminating each unfinished line with "&"; target-language statements (also continued using a final "&") are continued only if permitted within the target language syntax.

V. Compile-Time Features of the CRISP Processor

The CRISP processor has a minimal, but useful, compile-time text-macro capability. Target languages having better macro handlers may, therefore, choose not

¹The use of <* . . . * > to enclose comments is apt to be implementation-dependent; similar constructs, such as (* . . . *), or [* . . . *], or / * . . . * / may be advisable in some cases.

to have this particular feature implemented. There are two directives; the first is the *macro definition*,

%template MEANS *target string*%END

which declares that occurrences of the type *%source string* that match *%template* are to be replaced, both in CRISP control statements, as well as in target statements, by *target string*. An instance of the type *%source string* is an instance of a *macro call*. The *target string* may extend over many lines, defining a procedure and forming a block of text to be transferred; in such cases, each additional line is prefixed by %. The end of a defining macro is signalled by %END.

The macro template may contain formal parameters to be transmitted into the target string; these are signalled by the occurrence of the parameter marker in the template. Whenever a % occurs in an input source line, a scan of the remainder of the line initiates, much the same as in the STAGE2 macro processor (Ref. 14). When a match occurs between the input string calling macro and a macro template, the *target string* corresponding to that template is evaluated with the actual parameters resulting from the template match. The result of this evaluation replaces the matched source string in the output.

Correspondences between actual and formal parameters are set up during template matching. The template is a sequence of fixed strings separated by parameter markers (%), or "holes". When the matching process is complete, each parameter marker corresponds to some substring of the input line and the fixed strings exactly match the other substrings of the line. The *i*th parameter string gets inserted into the target string wherever occurrences of %*i* appear in *target string*.

Macro definitions and calls may be used anywhere in the CRISP source code; in particular, a call can precede the macro definition. Macro definitions may contain macro calls, but not other macro definitions.

The following CRISP program is an example of the use of the macro capability: Somewhere in the program, there is a definition module,

```
%RANDOM ARRAY MEANS A%END
%FILL %(:%) MEANS
%DIM %1(%2;%3)
%FOR DUM=%2 TO %3
%   %1(DUM)=RANDOM
%   NEXT DUM%END
```

The appearance elsewhere in the program of the call

```
%FILL %RANDOM ARRAY(1:50)
```

produces first the intermediate statements

```
DIM %RANDOM ARRAY(1:50)
FOR DUM=1 TO 50
  %RANDOM ARRAY(DUM)=RANDOM
NEXT DUM
```

which are then rescanned for CRISP control statements and possible further translations. In this particular case, there was further macro action, leading to the final CRISP code:

```
DIM A(1:50)
FOR DUM=1 TO 50
  A(DUM)=RANDOM
NEXT DUM
```

Each module can automatically be given a number by the CRISP processor in its annotation mode, and assigned a special module-entry counter to record the number of times that particular module has been executed when the program runs. The execution count for each module through level n can then be printed upon execution of the CRISP directive

```
DISPLAY THRU LEVEL  $n$ 
```

The value n is the level of hierarchical nesting within the program as determined by the decimal count in the Dewey-decimal module identifier, to be described a little later.

This path-execution-count capability is invaluable in program testing, for one may readily identify which paths have been executed and which have not. Moreover, because of the program structure, it is possible to design and provide input data to exercise these paths.

For fully verified programs, the overhead setting up and incrementing of these counters can be removed by prefixing the source program by the CRISP directive

```
CANCEL MODULE COUNT
```

Selected portions of a program may have their module counters enabled and disabled by using the directive

```
ENABLE MODULE COUNT
```

with the CANCEL directive above.

Perhaps the most unique of the compile-time features is what may be termed a "compile-time" edit statement:

```
REQUIRE AT  $m:s$ 
```

This statement causes the statement s to be inserted in the object code immediately before the code for module m , numbered as in the next section. Its purpose is to permit truly top-down development and readability of programs. For example, suppose a DO f appears inside a loop. At the time the DO f statement was written, the programmer envisioned that a certain *definite function* would be performed by an as-yet *undefined algorithm*. However, in programming the PROCEDURE: f at the next level, he may discover that, to program the intended function, an unforeseen variable needs to be declared and given an initial value back at an earlier program level, outside the loop.

But the program development up to this point was not concerned with this value. It has only just become important. Furthermore, the declaration and initialization of a *new* variable does not in any way alter the correctness assessment of the program up to that point (except perhaps in timing, if critical). Hence, it makes sense to associate the statement that initializes a procedure *with* that procedure, rather than back at the previous level. Otherwise, it threatens readability and understanding, both of the previous module ("what is *this* doing here?") as well as that needing it ("where on Earth did I initialize that variable, and what to?")

Every data structure need not be declared using a REQUIRE statement, some are naturally passed on to procedures as data on which they are to operate. Use of the REQUIRE, however, can enhance readability when internal structures need external initializations.

VI. Module Terminations

As I have discussed in another article (Ref. 5), there are times when module exits other than the normal structured exit are needed for program efficiency and clarity. These may take the form of responses to pathological or abnormal events, in which case, they are *abnormal terminations*. Sometimes, however, the event leading to a desired immediate nonnormal (nonstructured) exit is one that is expected. For example, it is a typical practice to

input data until an end-of-file indication signals the program to begin processing in a new mode. I call these non-structured exits from a module *paranormal* terminations (*para* from Greek meaning "beside").

```

IF NO  $t$  DURING  $s$ 
:-->THEN  $s_1$ 
:      :
:       $s_n$ 
:      END
:-->ELSE  $s_m$ 
:      :
:       $s_p$ 
:      ENDBLOCK

```

Here, s refers to a single statement, such as an input statement with t an end-of-file trap, or s may be a CRISP procedure call in which the traps, or t_1, \dots, t_k appear. The t 's can conceivably also refer to logical conditions tested during s . The vertical line of colons and ":->" represent flowlines, supplied by the CRISP processor in its annotation mode.

The CRISP directives that effect extra-normal exits from CRISP procedures are

EXIT t (or EXIT t TO m)

and

ABORT (or ABORT TO m , or ABORT l TO m)

in which t identifies the CASE-label (t may be null for exit to an ELSE-module) and m identifies the CASE- or ELSE- module number. "TO m " is optional on EXIT statements, and also on ABORT if the entire program has only one abnormal-termination procedure.

The CRISP processor restricts a module to having only one normal (structured) exit statement per module. How-

ever, there must have been a CASE- or ELSE-module identified at a previous level of the program to which such transfers are to occur. Two illustrative cases are

```

IF NO  $t_1, \dots, t_k$  DURING  $s$ 
:-->THEN  $s_1$ 
:      :
:       $s_n$ 
:      END
:-->CASE  $t_1:s_m$ 
:      :
:       $s_{m+1}$ 
:      :
:       $s_p$ 
:      END
:-->CASE  $t_k:s_r$ 
:      :
:       $s_r$ 
:      ENDBLOCK

```

ever, the top-down development of program modules having multiple exits may necessitate inserting several nonstructured exit statements into the module, and CRISP, therefore, allows them. However, these can sometimes create difficulty in isolating errors, because the program generally has no convenient way of telling *which* exit of the multiplicity was actuated.

The CRISP programmer should thus take care to use multiple extra-normal exits only whenever the point within the module returned to is insensitive to whichever of the multiple exits to that return point is taken.

VII. CRISP Module Numbering Method

Each CRISP block corresponds to a flowchart structure containing nodes and flowlines, and each CRISP statement either corresponds to a node or a flowline. One natural way of numbering graph nodes is the so called *pre-order traverse* method. A pre-order traverse of the chart enumerates the boxes on the flowchart as follows: starting at the top of a structured flowchart, label boxes and loop-collecting nodes sequentially in order down the chart until a branching node is sensed. Number this node. The general rule to be followed whenever a branching node

is reached is, take the leftmost unnumbered branch. Whenever a decision-collecting node is encountered, return to its corresponding decision node if it has a yet-unnumbered branch, and proceed to number the leftmost unnumbered branch; if all of its decisions have all branches numbered, then continue on.

One way that the hierarchic place a module occupies in a design can be annotated is by a Dewey-decimal cross-reference. For example, suppose that, on a flowchart numbered *m*, a box, numbered *n*, refers to a procedure (not subroutine), to be expanded later in the design process. Then the flowchart for that later expansion could be made Chart No. *m.n*. One reading the flowchart, wishing to trace out *how* the function in box *n* of flowchart *m* is achieved, merely has to locate Chart *m.n* to proceed.

More specifically, suppose a module appears on Chart 1.2.6, and has the number 5. Then one can state that box number 2 on Chart 1 was expanded as Chart 1.2; on that chart, box 6 was expanded as Chart 1.2.6; and module number 5 may appear expanded later as Chart 1.2.6.5.

This process can be altered for subroutines and functions also. The alteration is needed because subroutines, which can be called from many places, would not possess a unique chart number. Therefore, each subroutine will be assigned to its own unique level-1 chart number. One convenient way of distinguishing procedures from subroutines is by the use of an alphanumeric chart number; for example, S6 refers to Subroutine 6, and T4 to Trap routine 4, etc. The choice of an alphanumeric designator can be used to group subroutines with common properties together in documentation. Expansions within subroutine flowcharts follow the normal numbering, as, for example S6.4.2 refers to the box numbered 2 on Chart S6.4.

Comparing flowcharts and their CRISP code structures (see the Appendix) shows that when IF-THEN-ELSE configurations are drawn with *true* to the left of *false*, and when multiple decision branches always are drawn in case-order left to right, then the *code statements corresponding to numbered flowchart boxes always appear in the program in sequential numeric order from the top-down*.

The CRISP system, in its edit mode, can simulate the preorder traverse of flowchart nodes described above, and annotate certain lines of the code with its appropriate number. The format of this annotation for statements

within a procedures block defined by PROGRAM:, PROCEDURE:, SUBROUTINE:, FUNCTION:, or AT, is

.n statement

The *statement* can either be CRISP or target language code. The *.n*, flush with the left margin for easy identification, is the number assigned by the preorder traverse. Module numbers for PROGRAM:, PROCEDURE:, SUBROUTINE:, FUNCTION:, and AT statements are the Dewey-decimal reference numbers assigned earlier in the program; they appear flush at the right-hand margin, as

PROCEDURE: *name* MOD# *d*

Thus a statement *.n* within a procedure with Dewey-decimal number *d* is uniquely identified as the Dewey-decimal *d.n*.

Statements that invoke subroutines (GOSUB, CALL, and AT...) have module numbers of the form *.n/Ai*, which signals that module *n* of the current procedure calls the *i*th subroutine of a class with alphanumeric designation *A*.

VIII. Indentation and Annotation

Although the syntax does not require it, the program structured hierarchy should be displayed by indenting the lines of code, such as shown in the syntax table in the Appendix. Examples in this article are indented according to the following rule:

If a block contains only one module (such as a FOR-block), then indent statements comprising that module by a prespecified number of spaces (here, two) beyond the block header (the FOR). If a block contains more than one module (such as an IF-THEN-ELSE-block), then indent three spaces past the block header (IF) to the module header (THEN or ELSE), and each line of the module another five spaces beyond the module header. Certain one-module blocks also have a module header within the block, such as the IF-THEN block. These are indented as if they were multiple-module blocks for consistency. The particular numbers of spaces recommended above for indenting were chosen to aid the flowline annotation mode.

Programs indented this way are almost as easy to read as flowcharts, because the block type is identifiable by its

header, which protrudes from the body of the block, and the beginning of each module within the block stands out in the same way. Successive indentations occur for block structures within modules.

The CRISP processor, in its edit mode, supplies the necessary indentation and, in addition, annotates the code with flowlines and module numbers as shown in the following example:

```

PROGRAM:  BUBBLE-SORT
:      <* SORT IN-PLACE AND PRINT A SET OF NUMBERS
:      <* INPUT FROM A TERMINAL*>
:
.1  INPUT USING %PROMPTING MESSAGE AND FREE-FORM INPUT: N
:      DIM %ARRAY TO HOLD NUMBERS
:      PRINT 'ENTER NUMBERS TO BE SORTED:'
:      INPUT USING %FREE FORM: %ENTIRE ARRAY
:
.2  FOR N=N BY -1 TO 2 <*DROP OFF TOP ELEMENT EACH CYCLE*>
.3  ↑  FOR I=1 TO N-1 <*BUBBLE LARGEST ELEMENT TO ELEMENT N*>
.4  ↑  ↑  IF %ELEMENTS I AND I+1 OUT OF ORDER
.5  ↑  ↑  :->THEN %EXCHANGE VALUES
:      ↑  ↑  :----->ENDBLOCK
:      ↑  ←←NEXT I
:      ←←NEXT N
.6  PRINT \'SORTED VALUES:\'\'%ENTIRE ARRAY; <*BACKSLASH GIVES
:      <*CARRIAGE RETURN AND SEMICOLON CONTROLS SPACING WHILE
:      <*ARRAY IS BEING PRINTED*>
:--STOP

```

MOD# 1

```

<*MACRO DEFINITIONS:>
%PROMPTING MESSAGE AND FREE-FORM INPUT MEANS
  %'HOW MANY NUMBERS TO BE SORTED? #'%END
%ARRAY TO HOLD NUMBERS MEANS A(N)%END
%FREE FORM MEANS '(#)%END
%ENTIRE ARRAY MEANS A%END
%ELEMENTS I AND I+1 OUT OF ORDER MEANS A(I)>A(I+1)%END
%EXCHANGE VALUES MEANS A(I)=A(I+1)%END

```

The target language in the above example is MBASIC (Ref. 13), which performs exchanges via the operator ==.

It is certainly no more difficult to write structured-program code than it is to draw a flowchart, and both contain approximately the same level of detail. Some may argue, since the code listings have to be produced anyway, that supplying further documentation in the form of flowcharts is a duplication of effort. Moreover, maintaining consistency between human-drafted flowcharts and code listings during an iterative development cycle can be a very time-consuming task.

Furthermore, it can be argued that structured code is more rigorous than a flowchart. For one thing, it is written in a programming language whose syntax and semantics are well defined. For another, the structured code is part of the operating program, no translation being necessary (with its attendant possibility of introducing error).

Nevertheless, structured code, even with annotated flowlines (as in the CRISP example above), tends to be somewhat less graphic than a flowchart, and the rationale and functional specification of subprogram submodules

tends to be a little less understandable in code annotations than it is in the narrative which properly accompanies a flowchart.

Thus, while CRISP goes a long way toward illustrating *what* a program does very graphically as a self-documented product, it may not go quite far enough in communicating all the *whys* necessary for a reader to review and understand the program. I plan to show, in a later article, how

CRISP source programs can contribute to automatic flowcharting and automatic narrative documentation.

IX. Other Restrictions

CRISP must limit the use of some of its primitives within the syntax of the target language so that proper statement recognition is possible. The restrictions are as follows: target-language statements may not begin with the strings

'ABORT'	'ENABLE '	'IF '	'REQUIRE '
'AT '	'END'	'JOIN'	'RETURN'
'CALL '	'ENDBLOCK'	'LOOP:'	'STOP'
'CANCEL '	'EXIT'	'NEXT '	'SUBROUTINE '
'CASE '	'FOR '	'ON '	'SYSTEM'
'DISPLAY '	'FORK'	'PROGRAM:'	'THEN'
'DO '	'FUNCTION:'	'PROCEDURE:'	'UNLESS '
'ELSE'	'GOSUB '	'REPEAT'	'UNTIL '
			'WHILE '

As further restrictions, labels and subroutine names may not contain commas or colons, nor may the condition *c* of an IF, WHILE or UNTIL statement begin with the substring 'NO '. Further, the index *i* or value *v* for DO CASE may not contain ' OF '. When the OF form appears, the index *j* may not contain ' THRU '. The index *i* for ON may not contain ' DO ', ' GOSUB ', or ' CALL '.

In FOR-statements, the index *i* may not contain '='; n_1 may not contain ' BY '; and n_2 may not contain ' TO ', ' UNTIL ', or ' WHILE '.

The CRISP processor also requires that all values and labels in ON *i* and DO CASE statements must be defined as CASE labels within the ON *i* or DO CASE block. The ON *i* CALL (or GOSUB) statement (or both) may be absent if the target language does not allow calls by name or calls by label. Similarly, some of the other construc-

tions may not be implementable (in a useful form) for a particular target language.

If *t* is a trap list in an IF NO *t* DURING *s* statement, the ELSE module must be replaced by CASE modules corresponding to the trap identifiers.

X. Conclusion

This article has described a method and processor that extends the advantages of structured programming to arbitrary target languages. The use of CRISP facilitates and encourages top-down, hierarchical, modular development of structured programs, and contributes a large measure of self-documentation capability within the language constructs, annotations, and macro-expansion features.

References

1. Perlis, Alan J., "The Synthesis of Algorithmic Systems," *JACM*, Vol. 14, No. 1, pp. 1-9, Jan. 1967.
2. Böhm, C., and Jacopini, G., "Flow Diagrams, Turing Machines and Languages With Only Two Formation Rules," *Communications of ACM*, Vol. 9, pp. 366-371, 1966.
3. Dijkstra, E. W., "Structured Programming," *Software Engineering Techniques*, pp. 83-93. Edited by J. N. Burton and B. Randall. NATO Science Committee, 1969.
4. Mills, H. D., *Mathematical Foundations for Structured Programming*, IBM Document FSC72-6012. Federal Systems Division, IBM, Gaithersburg, Md., Feb. 1972.
5. Tausworthe, R. C., "Program Structures for Non-Proper Programs," in *The Deep Space Network Progress Report 42-21, March and April 1974*, pp. 69-81. Jet Propulsion Laboratory, Pasadena, Calif., June 15, 1974.
6. Hamilton, M., Zeldin, S., *Top-Down, Bottom-Up Structured Programming and Program Structuring*, Report E-2728. Charles Stark Draper Laboratory, MIT, Dec. 1972.
7. Dahl, O. J., and Hoare, C. A. R., "Hierarchical Program Structures," in *Structured Programming*. Academic Press, New York, 1972.
8. Brinch Hansen, P., *Operating System Principles*. Prentiss-Hall, Inc., New York, 1973.
9. Wulf, W. A., et al., *BLISS Reference Manual*, revised Oct. 25, 1971. Dept. of Computer Science, Carnegie-Mellon University, Jan. 15, 1970.
10. Miller, E. F., Jr., *A Compendium of Language Extensions to Support Structured Programming*, Report RN-42. General Research Corp., Santa Barbara, Calif., Jan. 1973.
11. Basili, V. R., *SIMPL-X, A Language for Writing Structured Programs*, National Technical Information Service Report AD755-703. U. S. Dept. of Commerce, Springfield, Virginia, Jan. 1973.
12. Flynn, J., *SFTRAN User's Guide*, Computing Memorandum #914-337. Jet Propulsion Laboratory, Pasadena, Calif., July 1973. (Internal document.)
13. *Fundamentals of MBASIC: Volumes I and II*, Jet Propulsion Laboratory, Pasadena, Calif., March and October 1973. (Internal document.)
14. Waite, W., *Implementing Software for Nonnumeric Applications*, Appendix A. Prentice-Hall, Inc., Englewood Cliffs, N. J., 1973.

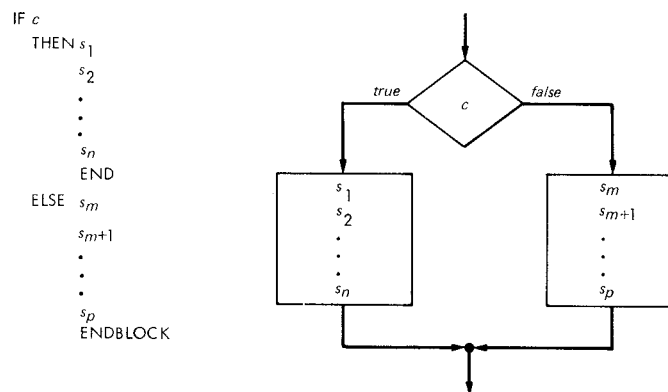


Fig. 1. The CRISP IF-THEN-ELSE structure

Appendix

Formats of CRISP Syntax and Flowchart Structures

This appendix identifies all the CRISP statements and indicates the pertinent context for the blocks and modules within blocks. *Italic elements in the listing below represent strings translated directly into either the target language syntax, or else strings to become CRISP procedure names and the like:*

c: condition, value corresponds to TRUE or FALSE convention

f: program, procedure, or subroutine name

i: index variable names in target language

j, h, k: integers

l: case or subroutine label

m: CRISP module number

n: numeric index quantifiers in target-language

s: statement, either target language or CRISP

t: trap, or event quantifier in target language

v: variable in target language

The flowchart structures for FOR-NEXT loops are shorthand conventions. The trap-handling conventions in ⑮–⑲ are discussed in Ref. 5. All subscripts in the table below represent integers.

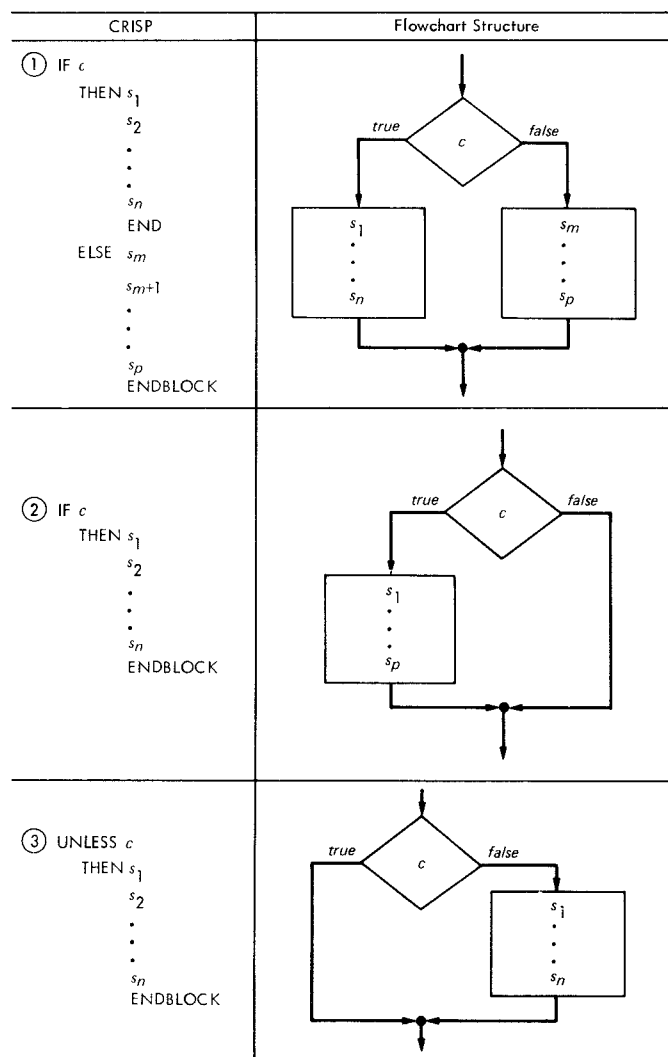


Fig. A-1. CRISP syntax and structure outline

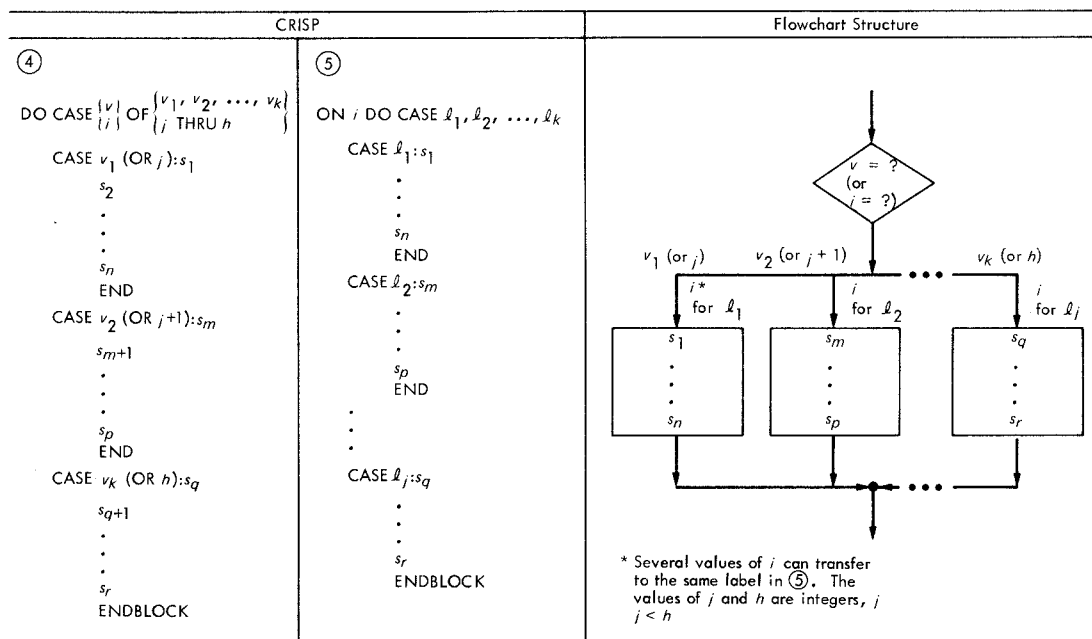


Fig. A-1. (contd)

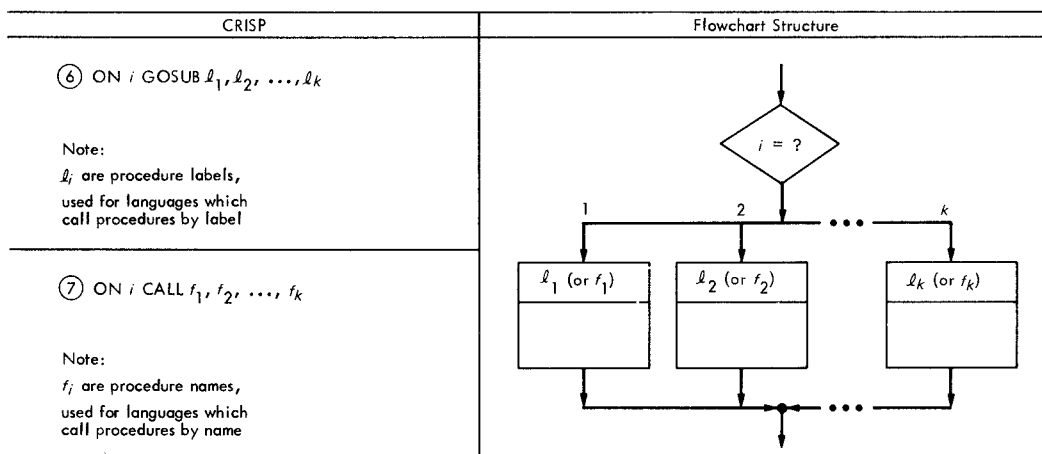
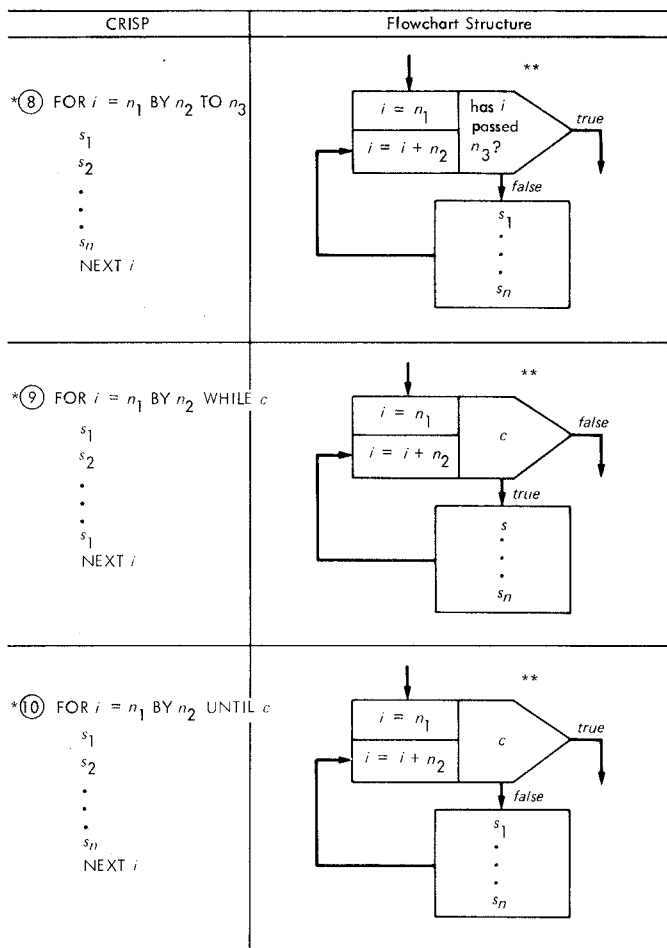


Fig. A-1. (contd)



* BY n_2 is optional on all FOR-blocks; BY 1 is assumed if omitted

** This flowchart symbol is a shorthand convention, merging initialization, decision, and update boxes together

Fig. A-1. (contd)

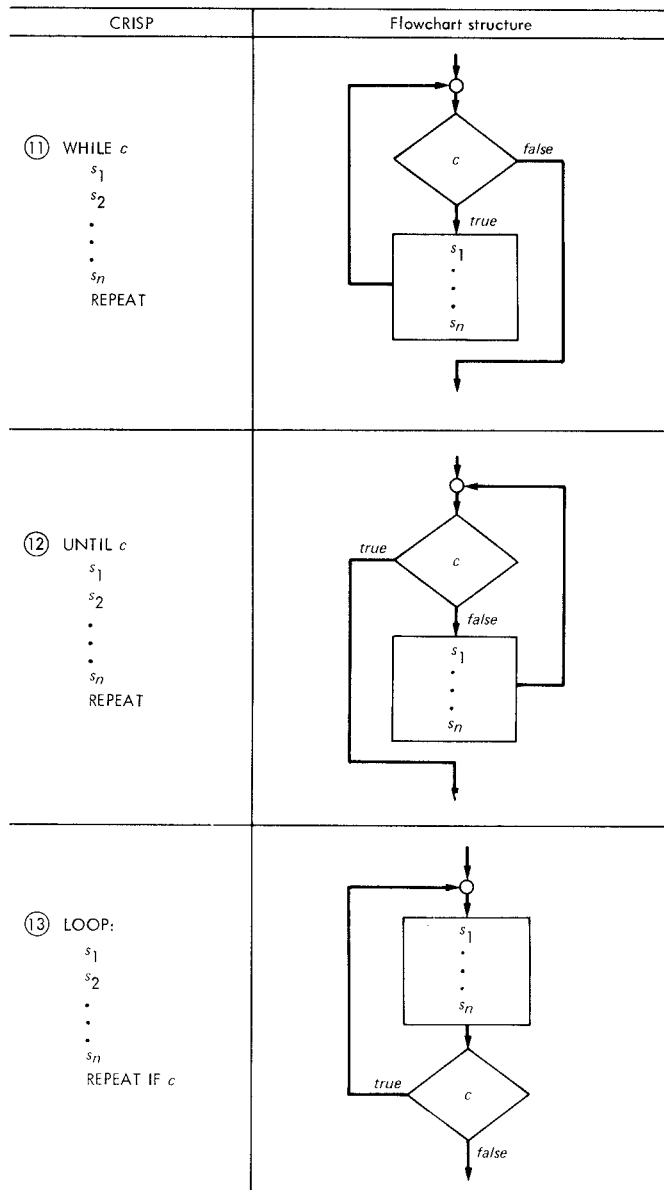
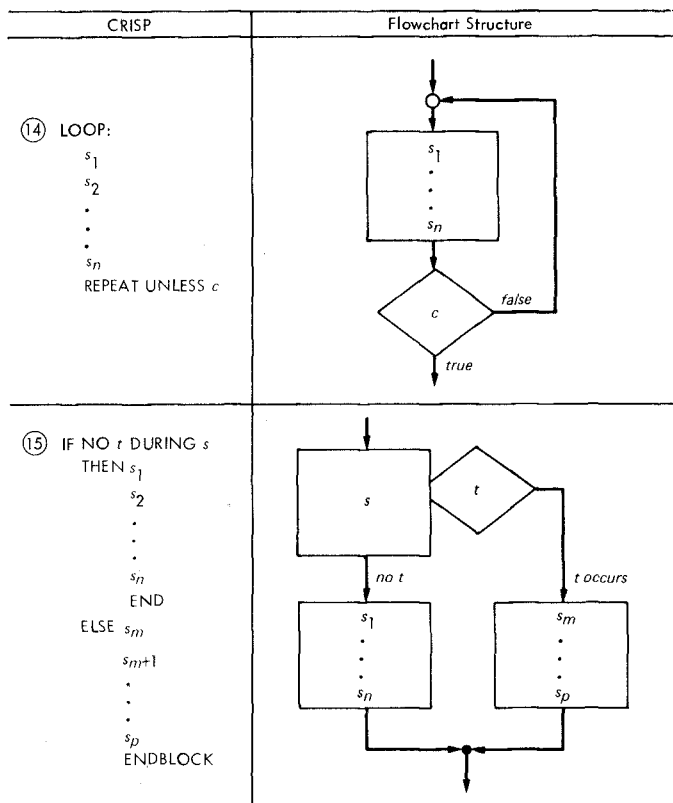


Fig. A-1. (contd)



Note:
 Multiple CASEs may replace
 ELSE module here if t
 during s results in multiple
 transfers. (See ⑤)

Fig. A-1. (contd)

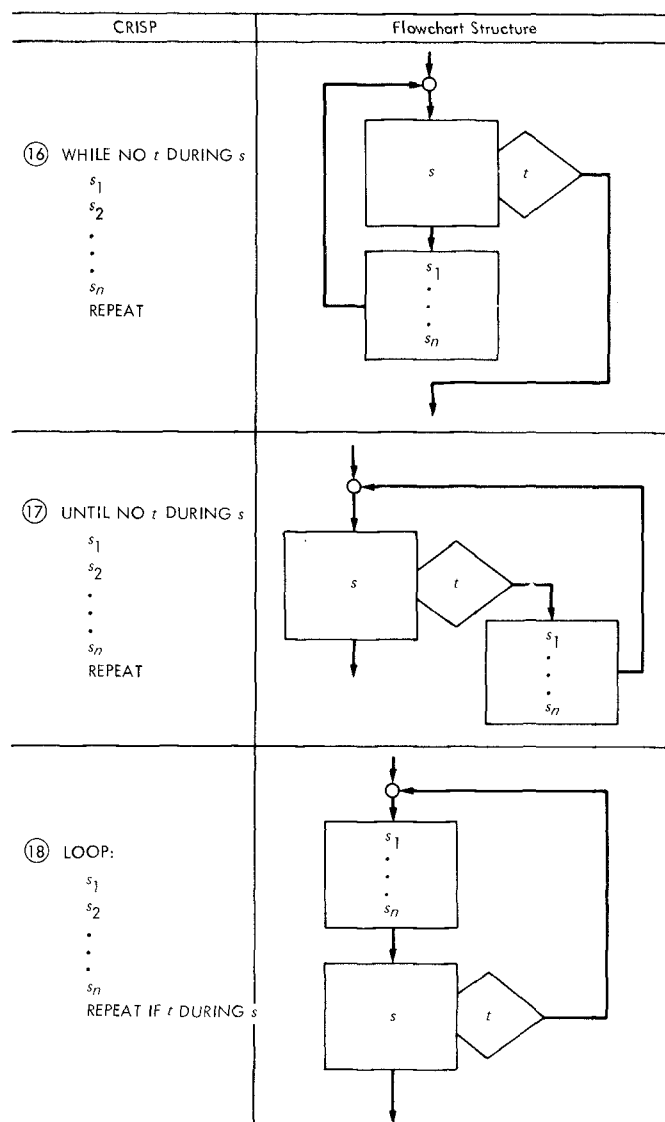


Fig. A-1. (contd)

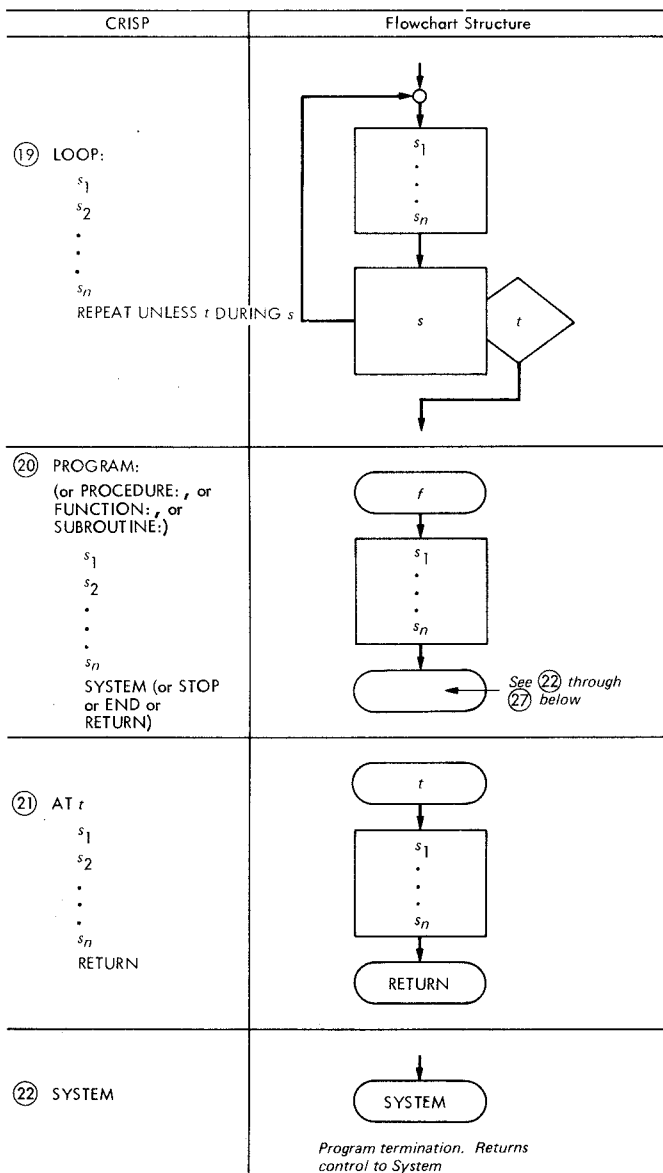


Fig. A-1. (contd)

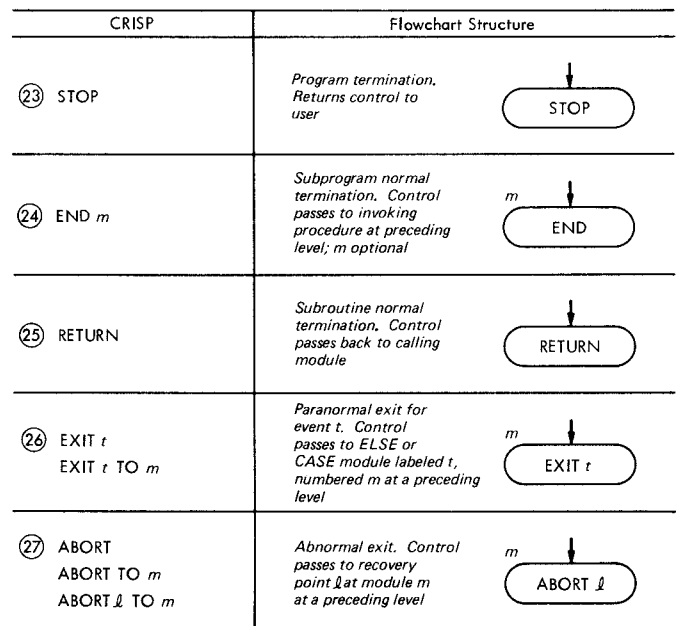


Fig. A-1. (contd)

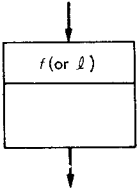
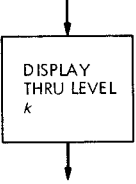
CRISP	Flowchart Structure
②⑧ DO f ②⑨ GOSUB l ③⑩ CALL f	
③① DISPLAY THRU LEVEL k	
③② ENABLE MODULE COUNT	CRISP processor directives: enable/disable module-counters
③③ CANCEL MODULE COUNT	
③④ REQUIRE AT $m:s$	CRISP processor directives: inserts statement s into code at the beginning of module m . Source must be annotated with module numbers
③⑤ %macro	CRISP processor directive. Creates or invokes a user-defined macro

Fig. A-1. (contd)

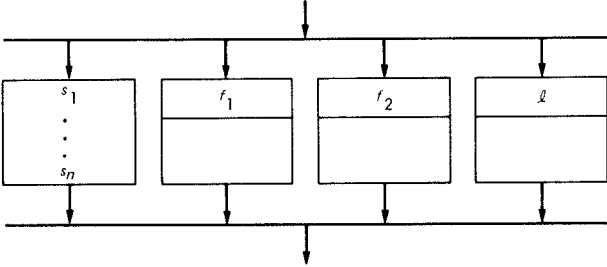
CRISP	Flowchart Structure
③⑥ FORK PROCEDURE: s_1 . . . s_n END DO f_1 CALL f_2 GOSUB l JOIN	

Fig. A-1. (contd)